Computer Engineering, Degree Project, Advanced Course,
15 higher education credits

Z EXCURSION – GAME ENTRY FOR SWEDISH GAME AWARDS

Tobias Nylin

Programme in Simulation and Computer Games Technology,
180 higher education credits

Örebro, Sweden, Spring 2012

Examiner: Lars Karlsson

Z EXCURSION – SPEL-BIDRAG TILL SWEDISH GAME AWARDS

Örebro universitet
Institutionen för naturvetenskap och teknik
701 82 Örebro

Örebro University
School of Science and Technology
SE-701 82 Örebro, Sweden

Abstract

This report describes my degree project. For my degree project I made a demo for a game idea that I've developed, and implemented for use with *Xbox 360*. The idea was about converting the feeling and gameplay of old 2D side-scrolling platform games into 3D, viewed from a first-person perspective. The Xbox 360 version of the demo was also submitted for the game competition *Swedish Game Awards*.

In the report, I explain this idea more extensive, what software I used to implement it, and how I implemented it. Finally, at the end I clarify the results and discuss fulfilled tasks and future improvements. I'm very pleased with my progress so far and I will continue to work on this on my own until it feels like a finished product that can be distributed.

Sammanfattning

Den här rapporten beskriver mitt examensarbete. För mitt examensarbete så gjorde jag en demoversion av en spelidé som jag utvecklat, och som implementerades för användning på *Xbox 360*. Idén bestod av att överföra känslan och spelmekaniken från gamla tvådimensionella sidoskrollande plattformsspel till 3D, sett ur ett förstapersonsperspektiv. Demoversionen för Xbox 360 skickades också in som ett bidrag till speltävlingen *Swedish Game Awards*.

I rapporten så förklarar jag idén mer utförligt, vad för mjukvara jag använde för att implementera den, och hur jag implementerade den. I slutet av rapporten så klargör jag resultaten och diskuterar mål som har uppfyllts samt idéer för att vidareutveckla spelet. Jag är nöjd med det jag åstadkommit hitintills och jag kommer att fortsätta arbeta på spelet själv tills det känns som en färdig produkt som kan distribueras.

Preface

I would like to thank my instructor Mathias Broxvall, for devoting about one to two hours per week for appointments with me. He was very enthusiastic in this task, and clarified a lot of doubts for me!

Contents

1. INTRODUCTION 4
   1.1. BACKGROUND 4
   1.2. PROJECT 4
   1.3. OBJECTIVE 4
   1.4. REQUIREMENTS 5
2. METHODS AND TOOLS 6
   2.1. METHODS 6
      2.1.1. What is XNA? 6
   2.2. TOOLS 7
      2.2.1. Textures and Level Design 7
      2.2.2. 3D Modeling in Blender 7
   2.3. OTHER RESOURCES 7
3. DESIGN & IMPLEMENTATION 8
   3.1. GAME CONCEPT 8
   3.2. CLASS STRUCTURE 8
      3.2.1. Object 8
      3.2.2. Resources 8
      3.2.3. Tickable 9
      3.2.4. Animated 9
      3.2.5. SkyBox 10
      3.2.6. Interactive 10
      3.2.7. Static 10
      3.2.8. Map 10
      3.2.9. Dynamic 11
      3.2.10. Platform 11
      3.2.11. Bullet 11
      3.2.12. Player 11
      3.2.13. Enemy 12
      3.2.14. World 12
      3.2.15. Audio 13
      3.2.16. Input 13
   3.3. DEVELOPMENT METHOD 13
   3.4. PRIMITIVE LEVEL EDITOR 13
   3.5. TEXT-FILE REPRESENTING ENEMIES AND PLATFORMS 14
   3.6. PHYSICS 15
      3.6.1. Equations of Motion 15
      3.6.2. Bounding-boxes and Collision Detection 16
      3.6.3. Response to Collision Detection 16
   3.7. MODELING PROCESS 17
   3.8. MUSIC AND SOUND EFFECTS 18
      3.8.1. XACT 18
      3.8.2. Playing Sounds in XNA 18
      3.8.3. Free Sound Effects and Own Sound Effects 18
   3.9. LICENSES 19
4. RESULT 21
   4.1. INSTALL 21
   4.2. START THE GAME 21
   4.3. HOW TO PLAY 21
5. DISCUSSION 24
   5.1. FULFILLED REQUIREMENTS 24
   5.2. GENERAL AIMS AND OBJECTIVES AND FOR THE PROGRAMME 24
   5.3. FUTURE IMPROVEMENTS 25
6. REFERENCES 26

1. Introduction

1.1. Background

I have done a game developed in *XNA* for use with *Xbox 360* and *Windows*. The genre of the game is a platformer viewed from a first-person perspective. A platformer is a game where the focus is on, jumping on, and between platforms, and also for avoiding obstacles. Platform games exist in both 2D and 3D.

The 2D platformers are usually viewed from the side and are referred to as *side-scrollers*, one example of a 2D platformer is *Super Mario Bros (NES)*. 3D platformers are often viewed from a third-person perspective and the player has more freedom than in the 2D counterparts. *Super Mario 64 (Nintendo 64)*, an example of a 3D-platformer, revolutionized the way platformers were played. Many 3D-platformes have taken inspiration from Super Mario 64 since then. Not least a later installment in the Mario-series called *Super Mario Galaxy (Wii)*. An early example of a 3D-platformer viewed from a first-person perspective is *Jumping Flash! (PlayStation)*. There also exists a sub-genre called *2.5D*, that merges the visuals and gameplay of 2D and 3D-platformers. 2.5D uses the gameplay and perspective of 2D side-scrollers, but is rendered in full 3D. As early as the mid-90's, 2.5D games started to show up, but they have appeared more frequently during the last decade. An example of a 2.5D game is *LittleBigPlanet (PS3)*.

The idea for the game developed as my degree project was to convert the gameplay and feeling of 2D side-scrollers viewed from a first-person perspective. Meaning, taking the concept of 2D levels and traversing them from a first-person perspective. I also wrote a concept and design document about this idea at another course called *Game Design*, at Örebro University, in 2008.

1.2. Project

The game consists of linear levels where the player can walk forward, backward, up and down. The latter two is mostly through jumping or falling by gravity. You can also aim at a certain degree with the right thumbstick on the Xbox 360 controller or with the mouse on Windows. This enables the player to get more variation when shooting at enemies. To turn around quickly there is a button for rotating the character 180°. This simplifies the gameplay since there isn't anything to interact with on the sides of the player, just in front and behind, and also to enhance the feeling of the initial idea.

1.3. Objective

The purpose of the project was to create something that feels unique, not entirely with focus on the technical part but rather something that is pleasing to play and is esthetic. The demo of the game was submitted to *Swedish Game Awards (SGA)* [1], which is a competition in game development for students resided in Sweden. SGA started in 2002 as the *KTH Game Awards*, KTH is the *Royal Institute of Technology*, based in Stockholm. SGA is a part of *Excitera*, which is a student-driven entrepreneurship association at KTH. Already at the beginning, prominent companies like *Microsoft* and *DICE* supported the competition. The second year the competition was held, another competition *Excitera Mobile Awards* was created, targeting mobile platforms. Though the following year, KTH Game Awards and Excitera Mobile Awards were merged into one competition, namingly SGA. This competition was aimed at both PC and mobile platforms. In 2005, the SGA-Tour was arranged for the first time, where the organizers travels to institutes all around Sweden to inform about the competition.

In the 2011-2012 edition of the competition (the edition in which this game participated), the following awards were listed: Game of the Year, Best Innovation, Best Mobile Game, Best Serious Game (educational), and Gamers Choice. The awards, with prizes, are also listed in *Table 1*. The available platforms for developing for was *Android*, arcade, *Flash*, *iPad*, *iPhone*, *Linux*, *Mac OS X*, *Nintendo DS*, *PlayStation 3*, *PlayStation Portable*, *Solaris*, *Wii*, Windows, *Windows Phone*, and Xbox 360.

The first event during this years SGA was the SGA-Tour. It already started in December, 2011, and ended in February, 2012. On January 18, the SGA-Tour visited Örebro University for a short briefing, which I attended. The second event was the *SGA Conference* held at KTH in Stockholm. Participants from all over the country gathered here to talk to each other, and to attend lectures from people inside the business. A similar event was also held in Malmö. The deadline for submitting the game was at May 31. To enter the competition a submission form was submitted at SGAs webpage. Also, the game itself along with a trailer and 3 screenshots were uploaded to an FTP server. The final event, at June 20, was the grand finale held at the Central Station in Stockholm, and an award ceremony at KTH.

1.4. Requirements

- Submission to Swedish Game Awards
- One playable level, at least 5 minutes of gameplay.
- Bitmap representing a level, for easy alterations of the level.
- Text-file representing a list of enemies with type, position and possible other attributes.
- Simple physics, movement of player, enemies, platforms and collision detection and response among these.
- Enemies as models.
- Textures or procedural textures for platforms.
- Artistic shading technique rather than state of the art.
- Infrastructure for music and sound effects.
- Sky-box.

Game of the Year	10,000 SEK
Best Innovation	5,000 SEK
Best Mobile Game	10,000 SEK and an invitation to ARM Cambridge office for private workshop with the Mali GPU engineers.
Best Serious Game	10,000 SEK
Gamers Choice	Eternal Glory.

Table 1: SGA Awards.

2. Methods and Tools

2.1. Methods

2.1.1. What is XNA?

XNA [2] was used for the development of the game. XNA is a collection of tools and a runtime environment that supports development of games and other interactive applications in C#. *XNA Game Studio* is an *IDE (Integrated Development Environment)* designed for working with XNA. It can be used to develop for Xbox 360 or Windows, for example. XNA Game Studio was first released on December 11, 2006. XNA was originally an acronym for *Xbox New Architecture*, but after Xbox 360 was released they changed it to *XNA is Not an Acronym*. The current release 4.0 was announced and released on March 9, 2010.

To develop for Xbox 360, the *.NET Compact Framework* is used. This is based on *.NET Framework* which consists of a *CLR* (*Common Language Runtime*; programs are executed on a virtual machine) and an extensive class library. Language interoperability is also supported, which means that code written in one language can be integrated with code written in another language. Furthermore, XNA is based on the .NET Compact Framework.

XNA consists of a large class library, which have been adjusted particularly for game development. A reason for this is to increase code reuse, so the game developers can focus more on the design rather than to build the entire engine from scratch. The library consists of namespaces. The *framework*, *audio*, *content*, *graphics*, and *input* namespaces were the ones that were frequently used in this project.

The framework-namespace can be thought of as the main-namespace, every other namespace in XNA is derived from this one. Three important classes in this namespace is *Game*, *GameTime*, and *GraphicsDeviceManager*. The Game class, is the main class of any XNA game. It consists of five important methods. The first one, *Initialize*, performs initialization of content that aren't graphics-related before the game starts to run. The second one is, *LoadContent*, in this method textures, models, audio etc. should be loaded and objects that depend on these should be initialized. The third one, *UnLoadContent*, is the opposite, here content should be unloaded. The fourth one is, *Update*, this is an important method where code for updating objects should be placed, this method is called once every frame. The final one is, *Draw*, also important and called once per frame, in this method code for objects that shall be drawn should be placed. The GameTime class of the framework-namespace is used for retrieving how much time that has passed since the last frame, by using the time retrieved it is possible to do calculations based on variable time step, which means that objects are updated consistent each frame no matter how much time has passed. Trough the GraphicsDeviceManager it is possible to change the resolution of the game or to switch to full-screen mode. The framework-namespace also contains some important structures. *BoundingBox* is a structure that is used for determining collisions between objects, this structure is explained more in section 3.6.2. *Matrix* and *Vectors* structures are frequently used for performing transformations of objects.

The audio-namespace contains classes for how audio should be handled, this is explained more in section 3.8. The content-namespace includes the class *ContentManager*, which is used for loading textures, models, and audio etc. The graphics-namespace is an extensive namespace that is used for handling, how to draw, and for drawing objects. It contains the *BasicEffect*, which is an alternative for using shaders. Instead of using external shader-files,

the code for illuminating the environment is defined directly in the C#-files. The *GraphicsDevice* is mostly used for drawing primitives, this can be achieved by using a vertex buffer and/or an index buffer, these two buffers are also declared in this namespace. The possible primitive-types that can be drawn are triangle-list, triangle-strip, line-list, or line-strip. Another class called *Model* is used for loading an external model-file and for displaying it on the screen. The *SpriteBatch*, is a 2D counterpart for the GraphicsDevice, it is used for displaying 2D graphics on the screen. The *SpriteFont*, is used for loading external fonts and are drawn by using a SpriteBatch. The *Texture2D* class is used for loading textures and by assigning texture-coordinates for objects vertices, textures can be mapped to them. Finally, different vertex formats can be used for telling the graphics card how to draw objects. The most basic one just contains position and color data. The one that is used in this project contains a position, normal and texture-coordinate.

The last frequently used namespace mentioned is the input-namespace. This is used for checking if the player has pushed any buttons on a gamepad or on the keyboard, or clicked a button on the mouse, or moved it. If any input is detected a different logic is executed depending on which button was pressed.

XNA is available for *Windows XP*, *Windows Vista*, *Windows 7*, *Windows Phone 7*, and Xbox 360. XNA supports easy portability across these platforms, if a game is developed on a computer with Windows it can easily be ported to e.g. Xbox 360. For distributing your game commercially to Xbox 360 a membership on *App Hub* is needed, the annual fee is $99 [3].

2.2. Tools

For the entire development process I used Windows, but for running the demo I used Xbox 360 as well. For the programming I used *Visual Studio* and XNA.

2.2.1. Textures and Level Design

For creating textures and designing levels I used *GIMP* [4] which is an acronym for *GNU Image Manipulation Program*. It is a free and open source raster graphics editor under the *General Public License (GPL)*, which means users have access to and can alter the source code. GIMP is mainly used for retouching photos, image composition (how to make a photo more attractive to the eye), and construction of images. It was the latter one I used for creating the textures. The maps of the levels are also textures. The textures were then imported in my project trough XNAs content-pipeline.

2.2.2. 3D Modeling in Blender

For the modeling part I used *Blender* [5], which is a 3D computer graphics software, free and open source under the General Public License (GPL). In Blender it is possible to make animated films, special effects for movies and even entire video games. For this purpose, I only used it to create primitive models with materials and exported them to the *DirectX* file format (with the extension *.x*). These were then imported trough XNAs content-pipeline.

2.3. Other Resources

For debugging the game on Xbox 360 it was required to own an *Xbox Live Gold Membership* and that *XNA Game Studio Connect* was installed on the console. [6]

3. Design & Implementation

3.1. Game Concept

The concept of the game was to do something creative with the old genre of side-scrolling platformers. Since this degree project was focusing more on programming, most time was spent on that. Less time was spent on developing a story and designing. At this moment, a story doesn't exist. However, for the design, retro-graphics was intended already from the beginning. The goal with the game is to go from point A to point B in a level and finishing a boss at the end. Perhaps there will be an option for choosing in which order these levels should be played. Regarding rewards, there will be checkpoints on the levels, and some form of artifact to retrieve after a finished level. Also, something to collect throughout the levels, thus motivating the player to collect these to get a better ending.

The player can move forward and backward, and also change direction. The player can jump and is affected by gravity. The player will fall if no solid ground is beneath. The player has an arm cannon attached to its arm. This arm cannon fires a small cannon ball and it has infinite ammo. In the level there are platforms that are either stationary or moving. There also exists an enemy-type depicting a robot. There are two variants of the robot. In the first one the robot can walk, fire, and turn at walls. However, if it walks out of an edge it will fall down. The second variant though, turns at edges as well. There is another type of enemy, a spike, which is stationary, if the player touches one of these, the player will lose a live immediately. The final enemy in the level, is a block of ice, it moves vertically between two platforms. This enemy can't fire. When it touches a platform above or below, it rests there for a while and then starts to move again, but in the opposite direction. Finally, there is a checkpoint during the level, and at the end there is a cube defining the goal.

3.2. Class Structure

The idea behind the particular class structure chosen was to focus on that it should be easy to extend and to reuse. In this section I will go trough the class structure in a top-down approach. A class diagram of this structure is illustrated in *Figure 1*.

3.2.1. Object

This is the base-class for all other classes in the program. Right now it doesn't contain anything of interest, except a print method and a type property that is accessed trough all its sub-classes. The type property returns the class name as a string of the instance calling it.

3.2.2. Resources

This is a singleton-class. A singleton is a class that can only have one instance, it is used for storing commonly used information that can be accessed by other objects in the system. Resources is a sub-class to Object. The main purpose of this class is to have a container where textures, models and audio are imported and stored. And also for the playback of the audio and for drawing textures and models.

It contains one instance of the following. The *ContentManager*, which is the pipeline for loading textures, models, etc. The *SpriteBatch*, that is used for drawing 2D-graphics on the screen. A *GraphicsDevice* for handling the drawing of polygons. The *BasicEffect*, which is a default shader in XNA, with restricted functionality but enough to illuminate the environment.

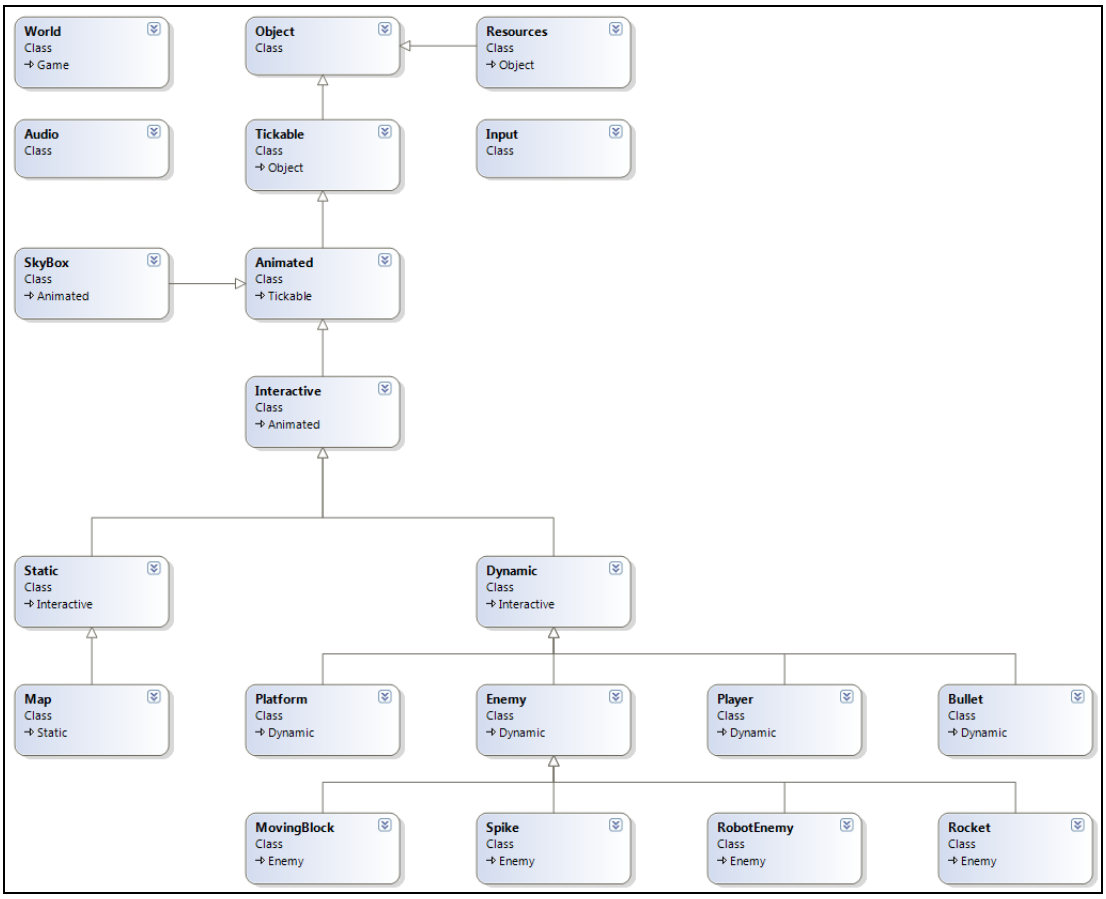

Figure 1: Class diagram.

It also contains instances of the pre-defined Model-class. For example, a model representing a particular enemy that was created in *Blender*, is imported into the project trough the content-pipeline and stored in an instance of the Model-class. Every time an instance of this enemy is drawn it uses the instance of the Model-class to draw it. In this way, if a model is instantiated more than once it still only has to be imported once.

The Resources-class also sets up a projection matrix and view matrix. Since the game is viewed from a first-person perspective the latter one changes depending on the player's position and orientation.

3.2.3. Tickable

This class contains a static list with all objects that depend on time. It has a static method for inserting objects into the list. The first time this method is called, it initializes the list. It also contains a virtual method for updating the objects, this method is overridden in the sub-classes, so that the correct update method is called for each sub-class. Tickable is a sub-class to Object.

3.2.4. Animated

Animated is a base-class for all classes that have objects that shall be drawn to the screen. There are three different type of objects that can be drawn. One type of object that can be drawn is just used for scenery and doesn't interact with other objects in the world. The second type are objects that can move and interact with other objects. The final type are objects that are stationary, which the objects that can move can interact with.

This class contains a vector for the position of objects which defines the point in space where the object shall be drawn from. Animated is a sub-class to Tickable.

3.2.5. SkyBox

This class is a sub-class to Animated and it contains a cube-map that is used for the skybox. A cube-map is a bitmap with six textures inside it, they are ordered like a flag with a cross, another way to see it is as an array with 3 rows and 4 columns. The middle row has 4 textures that lie next to each other, they could be labeled as: left, right, back and front texture. The top row has one texture representing the top of the cube, and the bottom also has one, but representing the bottom of the cube.

By assigning the correct texture-coordinates each of these 6 textures can be applied to one face each of a cube. But for a skybox the textures should be drawn on the inside of the cube. This can be fixed by temporarily changing the cull-mode to clockwise instead of counter-clockwise.

The skybox is then translated to the player's position and scaled. Every time the player moves the skybox will retrieve the player's new position, this means that no matter where the player is the skybox will always look as if it is at a constant distance, thus creating the illusion that it is very far away. The skybox is the object that is drawn first by all objects. It needs to be drawn with the depth buffer turned off, and therefore it will always be behind all other objects in the scene even though there might be other objects behind it, since it is not sorted depending on z-value. [7]

3.2.6. Interactive

This is a sub-class to Animated. The purpose of this class is to distinguish between animated objects that are just animated or that also interacts with the environment. For example, an explosion might be an animated object that doesn't need to interact with the other objects in the world, it could just be used for a nice effect when an enemy dies. On the other hand, an enemy is definitely an interactive object that needs to know what it collides with and how to respond.

This class contains a bounding-box and a vector for the relative bounding-box position. A bounding-box contains a min and max vector defining it's extremes. The min an max vectors of the bounding-box are calculated with this relative vector based on the current position of the object.

3.2.7. Static

This class is a sub-class to Interactive. It works as a base-class for classes that contain objects that don't move, i.e. they are static. For example, the cubes that the world is built by. The dynamic objects needs to know the cubes' boundaries so they can know when they have collided with a cube and how to respond to that.

3.2.8. Map

This is a sub-class to Static. The purpose of this class is to create cubes for each pixel's color-value in the bitmap representing the level that has been stated. A bounding-box is also created for each cube. If two or more bounding-boxes are adjacent horizontally they are combined into a single bounding-box.

The reason for this is that when the player falls down due to gravity, the player collides with a cube's bounding-box. If several cubes' bounding-boxes hasn't been merged, the player will

collide with the cube but not knowing how to respond, e.g. should the player move back up or should the player move left? This problem occurs because of the order how collisions are checked. First, I check if the player has collided with the right-face of a cube, then with a left-face, then with a top-face, and finally a bottom-face. Since I check the horizontal faces first it means that the player will think it has collided with a horizontal face and thus moves back or forward in the x-direction though it should move up in the y-direction. Therefore I merged all bounding-boxes that are adjacent horizontally.

When the entire bitmap has been iterated the vertices for all found cubes are stored in a vertex buffer. It is then used to draw all the cubes to the scene.

3.2.9. Dynamic

This class works as a base-class for classes that contain objects that can move, i.e. dynamic. It is in turn a sub-class to Interactive. It contains a vector representing velocity, that is used for moving the object based on variable time step, which means the final velocity is retrieved by multiplying the velocity by the time that has elapsed since the last frame. The reason for this is so that the objects will move proportionally against the time that has elapsed each frame.

Every dynamic-instance has an ID, and the class also has a static variable that keeps track of the next available ID. When a new dynamic-object is initialized it gets a unique ID. The class also has a variable for hit-points (health).

It has two virtual methods for how an instance should respond, one for other dynamic objects, and one for how to respond to the world. These methods are overridden in Dynamic's sub-classes. It also has a function that updates the bounding-boxes if necessary.

3.2.10. Platform

A dynamic class and thus a sub-class to Dynamic. This class represents platforms that can move. When calling this class' constructor you can determine if it should be a platform that moves in the vertical or horizontal direction, this is done using an enumeration. It's starting direction and how far it should travel before it changes direction should also be determined. For example, if a vertical platform with starting direction up and range 80 units is initialized, it moves 80 units up in the vertical direction, than changes direction to down and moves 80 units down, and then changes to up again, and so on.

3.2.11. Bullet

A sub-class to Dynamic. An instance of this class is only created if the player or an enemy has fired its weapon. Then an instance is created relative to the trigger's weapon. If the player is the trigger it also needs to know in which direction the player's weapon is facing, since the player can move the weapon for aiming. An instance is removed if it has collided with any dynamic or static object, or if it has left the game area.

3.2.12. Player

This class is the most extensive class. It contains all the logic for the movement of the player. It is a sub-class to Dynamic. Unlike other objects in the world the player itself isn't a model, the player views the world from a first-person perspective, this is the same as the camera's position, which is defined by the view matrix. Based on the player's movement and rotation the view matrix is updated.

For defining a view matrix, a position, target and up vector is required. In this case, position is the position of the player, the target is in which direction the player is looking and the up

vector is the positive y-axis, since the player cannot rotate freely it is enough to have the positive y-axis as the up vector. Furthermore, to define the target vector another vector called heading is used. This is a normalized vector, meaning the length of its components is always 1. The heading vector defines the direction in which the player is facing right now. If the player rotates, a matrix is created with the current angle, and is used to transform the heading vector. Finally, to get the final target vector the player's position and heading vector are summed. [8]

The biggest method in the player class is the update method. The first thing that is checked here is if the player has done any input. That is, pushed any buttons or moved the thumbsticks on the Xbox 360 controller, or pressed any keys or moved the mouse. Depending on the input, different actions are performed. Mainly the actions are based on movement and rotation, and also if the player has triggered its weapon.

While the player itself isn't a model, it has a model picturing an arm cannon (weapon), this model is placed relative to the player's position so when the player moves the arm cannon moves indirectly. It is also rotated based on the player's pitch and yaw angle. A pitch rotation defines a rotation vertically, imagine an airplane that ascends or descends. A yaw angle on the other hand defines a rotation that is horizontal, imagine spinning yourself around.

To translate and rotate the arm cannon correctly, a side vector perpendicular from the heading and up vector is created. This is done by using the cross-product. The side vector is then normalized. The player's position is then stored in a vector defining the basis for the arm cannon's position. Next, each component of this vector is transformed with the heading, up, and side vector. All these three vectors are normalized. Yet another vector, though constant, is used for defining the relative offset for the arm cannon. By multiplying the heading vector with the x-component of the offset vector, the up vector with the y-component of the offset vector, and the side vector with the z-component of the offset vector, and adding these three to the player's position the correct relative position of the arm cannon is retrieved. Since the heading, up, and side vectors are normalized, the relative distance from the player's position will always be the same. Finally, the arm cannon's transformation matrix is defined by first translating the arm cannon to the newly calculated position, and rotating it with the yaw and respectively the pitch angle, and lastly scaling it. [9]

3.2.13. Enemy

This class serves as a base-class for all different enemy types in the game. It is also a sub-class to Dynamic. It contains some protected methods for different behaviors, e.g. walking, falling and shooting. These can all be accessed in sub-classes to Enemy, so for example, for every enemy type that shall walk, they can just call the base-class' walk method from their update method, and they will walk. Every sub-class to Enemy can contain their own specific data that is not general for all enemy types. [10]

3.2.14. World

This is the main class of the game, this is where all the objects are instantiated, updated and drawn. In this class' update method all objects are first updated and after that they are checked against each other if they have collided. The objects that have collided performs corresponding response depending on the object they've collided with. If some objects have been marked for removal, these objects are removed at the end of this method.

3.2.15. Audio

This class handles the playback of audio in the game. It has methods for playing, pausing and stopping sounds. To start playing a sound, the name of the sound to play is given to the play method. The class also has an update method. The instance of this class is a member of the Resources-instance, and therefore it is accessible throughout the system.

3.2.16. Input

This class is a standalone class and checks for input, particularly from the gamepad, but also from the keyboard and the mouse. Built in structures in XNA are defined for checking events among these input devices. There is only one instance of the Input class, which is accessible throughout the system.

3.3. Development Method

Because of the lack of game programming experience before this project, it was hard to know exactly how to structure things in a good way. Meetings with the instructor occurred once a week throughout the whole project. Actually, the meetings began two weeks before the project officially started. At those two meetings we started to sketch on a class-structure. The concept we had in mind was that it in the end should be easy to add new elements to the game and that a lot of code should be reusable. At these two meetings we also formed a plan for things that should be implemented each week throughout the project.

Overall, the development method consisted of that we first met early in the weeks, usually Mondays, and discussed problems that had occurred the previous week and discussed more in depth about what should be implemented the current week. The meetings was also a good place to get feedback on things that had been accomplished so far. During the implementation itself, the experience from the meetings were useful, mostly it meant that the implementation was going on well. For some parts though, particularly the response for collisions, some trial and error had to been forced. The collision response was scratched two times before a good and efficient method was found. It should also be mentioned, that the collision response method isn't perfect, there is still room for improvement. To summarize the implementation method, mostly it was about implementing things in a logical order. First, the world was created, then the objects, after that movement of objects, then collisions between objects, and finally response on these collisions. Before a new part was implemented the previous part was tested carefully to really see that it worked as planned.

3.4. Primitive Level Editor

For designing a level I used a bitmap. Using a text-file for enemies didn't work on Xbox 360, therefore enemies were also added in the bitmap for the Xbox 360 version. A disadvantage with this was that all enemies of one type had to be identical, because for each color-value every argument has to be constant. The bitmap was designed in GIMP where different color-values represent different objects. For example, an object with an ice texture has a blue color, while a certain enemy type has a red color. The levels are designed from the side, much like the perspective used in side-scrolling games, but in the game the perspective is viewed from first-person. This is shown in *Figure 2*.

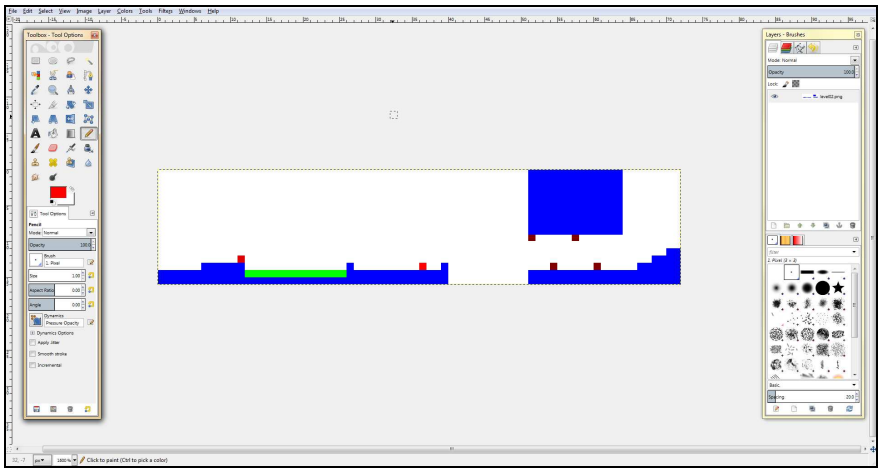

Figure 2: Level design in GIMP.

3.5. Text-file Representing Enemies and Platforms

A text-file is used in a way so that it is easy to add, remove or alter dynamic objects. The syntax of this file can be seen in *Figure 3*. Two different dynamic objects can be added in the text-file, an enemy or a platform. However, the former one is a base-class for all enemy types, so it is possible to create several types of enemies in the text-file. The text-file were only used on the Windows version, since the Xbox 360 couldn't find and handle it.

The text-file is read when the game executes by the use of a *StreamReader* instance. The StreamReader-class reads text-files. From the stream, one line is read at the time and each word or number is inserted into an array. This is done by telling the stream-reader to group characters together as long as no space-sign is encountered. Depending on the data collected it selects the proper class to create an instance, and insert it into the list of dynamic objects.

Syntax for enemies:
[class-name] [x] [y] [z] [speed] [start-direction]

Syntax for platforms:
[class-name] [x] [y] [z] [speed] [range] [start-direction] [type]

Legend:

[class-name]:	the name of the class that should be instantiated.
[x]:	grid-position on the x-axis.
[y]:	grid-position on the y-axis.
[z]:	grid-position on the z-axis.
[speed]:	the speed in which an object will travel at.
[range]:	the distance a platform will travel before changing direction, given in units not grid-position.
[type]:	defines if a platform should move horizontally or vertically.

Example:
RobotEnemy 4 3 0 4 Forward
Platform 12 3 0 4 80 Forward Horizontal

Figure 3: Syntax for creating enemies or platforms in a text-file.

3.6. Physics

3.6.1. Equations of Motion

For the motion of objects I used the *equations of motion* [11]. These are only working with constant acceleration and when the motion is along a straight-line. Constant acceleration means that the rate of change in velocity is uniform. Also, when acceleration is constant, changes in velocity is proportional to time. In addition, the displacement is proportional to both velocity and time. The motion of an object depends on three kinematic variables: velocity, displacement, and time.

In total, there are 4 different equations of motion, one for calculating the new velocity (Eq. 1), one for calculating the new displacement (Eq. 2), these were the only two that were used. The remaining two equations of motion are both combinations of the two which were used.

$$\textit{(Eq. 1)} \quad v = v_0 + a\Delta t$$
$$\textit{(Eq. 2)} \quad x = x_0 + v_0\Delta t + \tfrac{1}{2}\, a_x\Delta t^2$$

The only acceleration that was used, is the acceleration due to gravity, which is a constant acceleration, for real this is approximately 9.81 m/s^2 on Earth. Although, it is not a number that worked well in the game, a gravity of -3.00 units per second squared was chosen instead. The gravity is negative because the negative y-axis goes in the direction beneath the world.

For updating the position vertically (Eq. 2) wasn't used right away. Instead the velocity was calculated using (Eq. 3), which in theory is the same as (Eq. 1), the apparent difference is that it is specified for vertical velocity, and that the acceleration has been replaced by a *g* which stands for gravity. After that a modified version of (Eq. 2) was used, which is defined in (Eq. 4), the difference is that the acceleration part of the equation is removed because the velocity had already been updated by (Eq. 3). The reason why it was divided into two equations was because a threshold was used for the velocity, so when the velocity reaches a pre-defined max value it will stay on that velocity until the velocity starts to change in the opposite direction.

$$\textit{(Eq. 3)} \quad v_y = v_{0y} + g\Delta t$$
$$\textit{(Eq. 4)} \quad y = y_0 + v_y\Delta t$$
$$\textit{(Eq. 5)} \quad x = x_0 + v_x\Delta t$$

If the player decides to jump, the velocity is set to a constant positive scalar representing the initial jumping velocity. The player starts to jump with this velocity, but for every frame, gravity affects this velocity, after some time the velocity becomes less than zero and the player starts falling down again. The player continues to fall until it has collided with any static object or moving platform in the world.

The horizontal velocity is constant, so it doesn't need any acceleration at all (Eq. 5). Although, when playing with an Xbox 360 controller the left thumbstick is used for moving forward or backward. By pressing up on the stick the player moves forward, and by pressing down it moves backward. The thumbstick returns a decimal fraction value between -1 and 1, depending on the direction or how much the player has tilted it. This value is multiplied with the horizontal velocity of the player before the new position is stored. The result of this is that the player will move slower the less the player has tilted the thumbstick.

The equations of motion are also used for moving enemies and platforms. Enemies' motion is a similar to the player's motion, except that they cannot jump. They only move vertically and fall due to gravity. Platforms aren't affected by gravity, they either move vertically or horizontally.

3.6.2. Bounding-boxes and Collision Detection

The Interactive-class has a bounding-box member which can be accessed by all sub-classes to it. Every sub-class can add a bounding-box, representing the boundaries for the particular model that is used for that class. For defining a bounding-box the built-in XNA structure *BoundingBox* was used. This structure contains two vectors. The first one, *Min*, defines the minimum coordinates of the bounding-box, the second one, *Max*, defines the maximum coordinates.

One way for checking if a collision has occurred is by creating a union by the two bounding-boxes' min and max vectors. To get the min union vector, the method *Max* is used, this method returns the maximum value of each component pair in the vectors. For the max union vector, it is similar but the method *Min* is used instead. After this, these two union vectors are used to check if a collision has occurred, this is given in *Figure 4*.

Although I'm not using this approach since there is a method called *Intersects* in the *BoundingBox* structure, but I'm guessing it's using a similar approach internally. This method takes one parameter in the form of another bounding-box and returns true if this instance's bounding-box and the parameter's bounding-box has collided.

3.6.3. Response to Collision Detection

After a collision has been detected between two objects correct logic must be chosen for how these two objects should respond. There are two different main objects in the game, static and dynamic objects. Static objects are stationary, while dynamic objects can move. If a dynamic object collides with a static object, only the dynamic object needs to respond. On the other hand, if two dynamic objects collide with each other, both have to respond, mostly.

Because there exists both static and dynamic objects, the dynamic objects needs to have two separate functions, one for handling response to the world (i.e. static objects), and one for handling response to other dynamic objects. These methods are defined as virtual methods in the Dynamic-class, these are then overridden in each of Dynamic's sub-classes, thus making response-logic individual for each different dynamic object type.

Perhaps the most trivial response-logic is between a bullet and an enemy. If these two dynamic objects collide, the bullet is removed from the game and the enemy loses a certain amount of health.

```
BoundingBox bb1, bb2;
Vector3 min = Max(bb1.min, bb2.min);
Vector3 max = Min(bb1.max, bb2.max);

if (min.x < max.x || min.y < max.y || min.z < max.z)
        return true;
else
        return false;
```

Figure 4: Pseudo-code for collision detection.

A more complex response-logic is between the player and the world. The player can approach four faces of a cuboid. Depending on which face the player has collided with, different logic needs to be used.

For example, if the player has collided with the top (the ground) of a cuboid, the player needs to move back in the vertical direction, so it appears as the player is standing on the ground. To really know that the player has collided with the top of a cuboid, the distance between the min vector's y-component of the player's bounding-box and the max vector's y-component of the cuboid's bounding-box is calculated. Then, the absolute-value of this distance is checked if it is smaller than a certain range, which defines the top of the cuboid. If this is true and the player is also falling down (i.e. it's vertical velocity is below zero), it means that this is the face the player has collided with. To move the player back so it doesn't collide, a small number is used ε, this number is multiplied with the vertical velocity and added to the position of the player, then it checks if there still is a collision, if it is, ε is used again until the player isn't colliding with the cuboid anymore. This is given in *Figure 5*.

The remaining three faces of a cuboid that the player can collide with uses a similar response, it check differences between extremes and depending on the direction of the velocity, different responses are chosen. They also use an ε for moving the player's position back.

3.7. Modeling Process

The focus in this project was not to make elegant models, therefore much time wasn't spent on designing models. But as an easy and quick compromise, a model of a robot consisting of only cubes was created. Even the eyes of the robot are cubes, they are translated a tiny fraction so that they appear in front of the head. This model only has basic materials. A screenshot of the Blender interface and the Robot-model is shown in *Figure 6*.

Another model, illustrating a spike, uses textures. This model is represented by a square pyramid, meaning that it has four sides. Each of these sides is a triangle, to apply a square texture to these the correct UV coordinates had to be given. UV coordinates are used to define the part of a texture that should be applied to a face. U is the horizontal coordinate of the texture, while V is the vertical. UV coordinates are usually defined in the range from 0 to 1. For defining the part of the texture I wanted to use for a face of my pyramid, I had to give three UV coordinates. One at the bottom-left of the texture, one at the bottom-right, and finally one at the middle at the top, thus forming a triangle in the texture.

The model and its associated texture where then imported in XNA and linked correctly so the model knew the relative path to the texture.

```
if (abs(this.min.y - cube.max.y) < range.y && this.vel.y < 0)
{
        do
        {
                this.y += epsilon * this.vel.y;
        }
        while (this.Intersects(cube);
}
```

Figure 5: Pseudo-code for collision response.

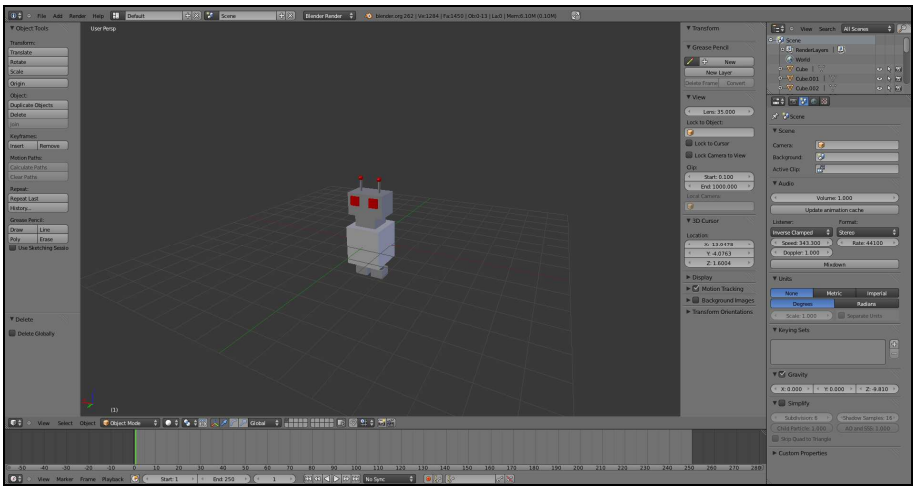

Figure 6: Modeling in Blender.

3.8. Music and Sound Effects

3.8.1. XACT

Importing sound effects and music isn't done in the same way trough the content-pipeline like models and textures. Instead, an external program called *Microsoft Cross-Platform Audio Creation Tool* is used, or for short *XACT* [12]. A screenshot of the XACTs interface is shown in *Figure 7*.

To use XACT a new project is first defined. Then a *Wave Bank* and *Sound Bank* is created in the project. In the Wave Bank, wave files are imported. The Sound Bank consists of sounds and cues. By dragging a wave file from the Wave Bank and drop it in the Cue-section of the Sound Bank, XACT automatically create a sound name and cue name for the wave file. Trough the sound name it is possible to modify the sound, e.g. altering volume and pitch, enabling looping and applying filters. In this way the sound could be altered without having to open it in an external audio editing software. The cue names are used for playing the sound from XNA.

3.8.2. Playing Sounds in XNA

First, the project created in XACT is loaded into XNA trough the content-pipeline. Then, for being able to play sounds an instance each of *AudioEngine*, *WaveBank*, *SoundBank* and *Cue* is needed. The AudioEngine instance is the instance where the project from XACT is loaded into. The WaveBank and SoundBank instance are then linked to the Wave Bank and Sound Bank that were created in XACT, correspondingly. In this way, it is not needed to write code every time a sound should be added or removed. This is done in XACT by dragging or removing sounds. Finally, for playing sounds, the Cue instance is used. It could also be used for pausing or stopping sounds.

3.8.3. Free Sound Effects and Own Sound Effects

I have used sound effects that I've downloaded from the Internet and also some that I have made on my own. When downloading sound effects I had to think about that they need to be free for commercial use. Right now the game isn't in a state for going commercial, but it might be a good habit to have sound effects that are properly licensed. Some of these sound effects was edited with *Audacity* [13], which is a free and open source sound editing software. The interface of Audacity is shown in *Figure 8*.

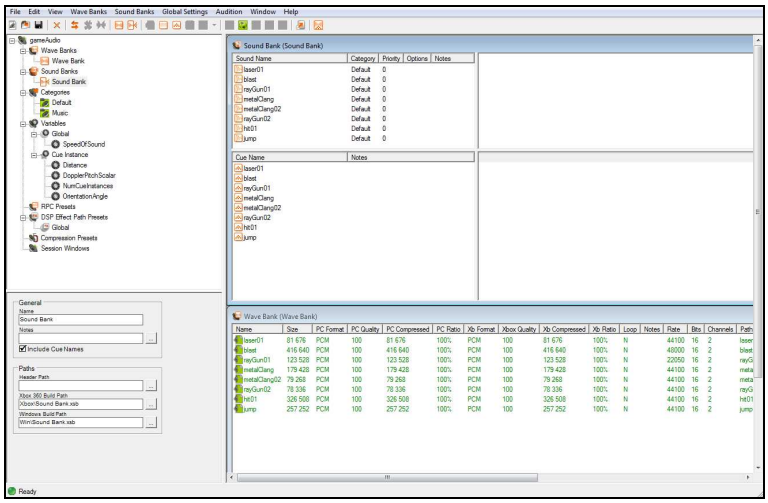

Figure 7: XACT interface.

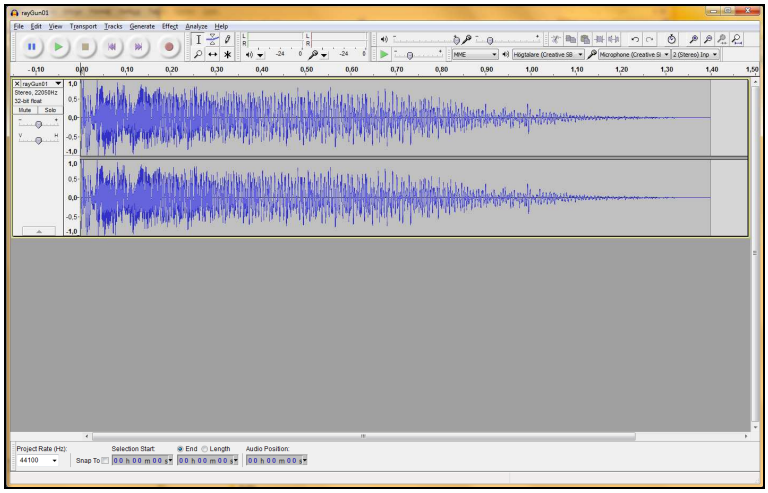

Figure 8: Audacity interface.

For the sound effects I did on my own, I used *FL Studio* [14], a *Digital Audio Workstation (DAW)*. In FL Studio I used a *VSTi (Virtual Studio Technology Instrument)*, called *z3ta+*. For example, I created a sound for use when the player jumps. I wanted it to sound retro. I searched for a proper preset in z3ta+, then at the beginning of the sound I pitched it up, and at the middle of the sound I pitch it down again, thus creating a sound that sounds similar to old platform games. The interface of FL Studio is shown in *Figure 9*.

3.9. Licenses

I have strived to use content in the game that I either have a license to use or that are free for commercial use. I've used GIMP for texture creation and level design, and Blender for modeling. For sound effects I've created on my own, I've only used software which I have licenses for. Moreover, for sound effects I've downloaded I used sounds that are licensed under the *Attribution 3.0* license [15]. Sounds under this license are free for commercial use, and to share, and to edit. Although, a certain sound that is used in a work needs to be attributed if the author requires that. Attribution 3.0 is a license under *Creative Commons* [16], on their webpage they state their mission like this:

> Our vision is nothing less than realizing the full potential of the Internet — universal access to research and education, full participation in culture — to drive a new era of development, growth, and productivity.

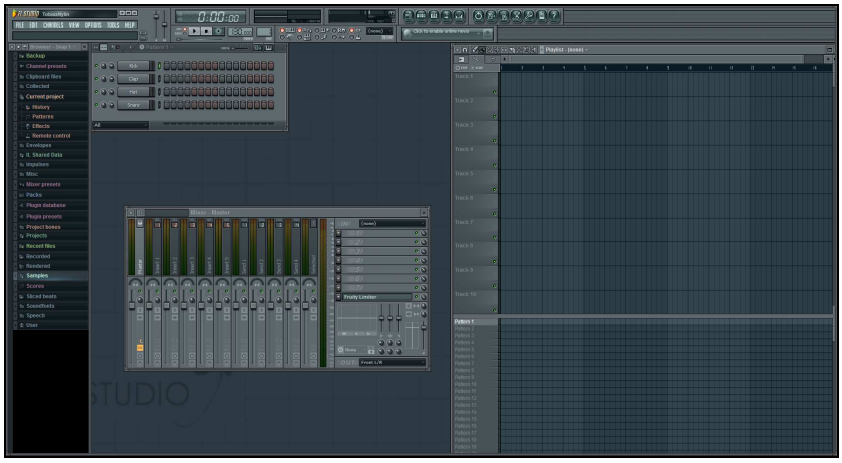

Figure 9: FL Studio interface.

4. Result

The result of this project is a demo-level of about 5 minutes of gameplay. The demo is played on Xbox 360. Windows was mostly used for debugging. The game was submitted for Swedish Game Awards but wasn't nominated in any category. Although, I wasn't taking that so hard. On the opposite, I was surprised how much I could achieve on my own during these 10 weeks only. A lot of the other participants worked in teams and had worked on their game for more than 10 weeks. If I had had just 10 weeks more I think I could have had a better chance.

4.1. Install

1. Start *XNA Game Studio Connect* on your Xbox 360. (make sure your computer is properly connected to your Xbox 360)
2. Double-click the "Z Excursion.ccgame" file on your computer.
3. *XNA Game Studio Package Utility* starts.
    a. make sure the platform to unpack to is your Xbox 360.
    b. Click on *unpack*!

4.2. Start the Game

1. On your Xbox 360, select *games*, and press A.
2. Select *My Games*, and press A.
3. Select *Z Excursion*, and press X to launch!

4.3. How to Play

When the level has started you can move forward and backward while tilting the left thumbstick up and down. To aim with the arm cannon, the right thumbstick is tilted in any direction, however, it is only possible to aim in a certain range. To jump, the left-bumper-button is pressed, and to shoot the right-bumper-button is pressed. For rotating 180 degrees, the right-trigger is pressed. The control scheme is also illustrated in *Figure 10*.

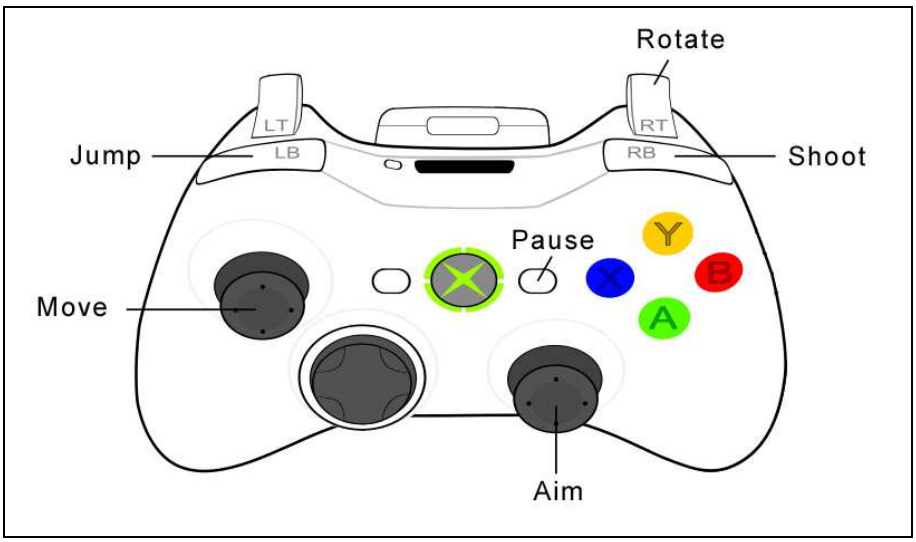

Figure 10: Control scheme.

The purpose of the demo-level is to get to the end of the level. The player starts with 5 lives. Every time a player loses all its health, or falling down a pit, or stepping on a spike, the player loses a life. To keep track of how much health the player has left, a life-bar is displayed at the top-left corner of the game interface, this is shown in *Figure 11*, *Figure 12*, and *Figure 13*. If the player has lost all his lives, the player gets game over and the game is moved to the game over screen. Here, the player can press the back-button to return to the title screen or the start-button to try the demo-level again.

If the player gets to the end of the level without getting game over, the game is moved to the level clear screen. Here the player can press the back-button to return to the title screen.

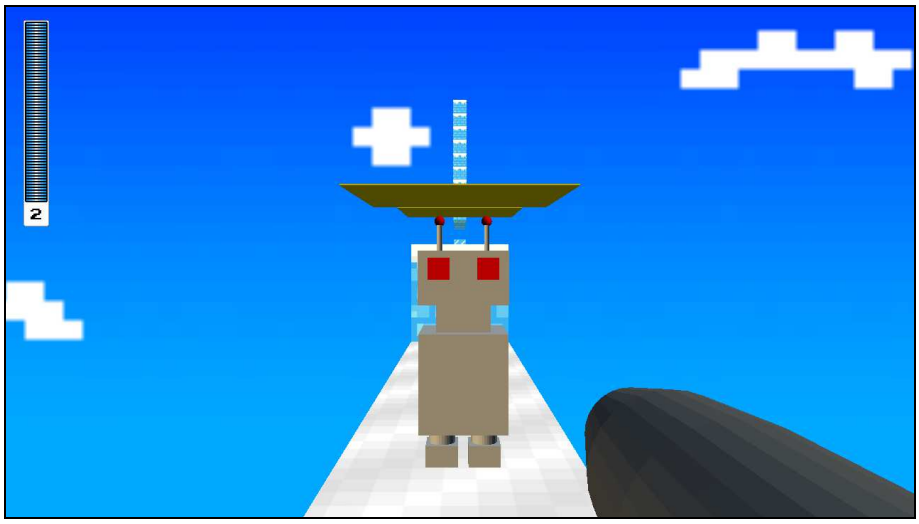

Figure 11: Start position of the level.

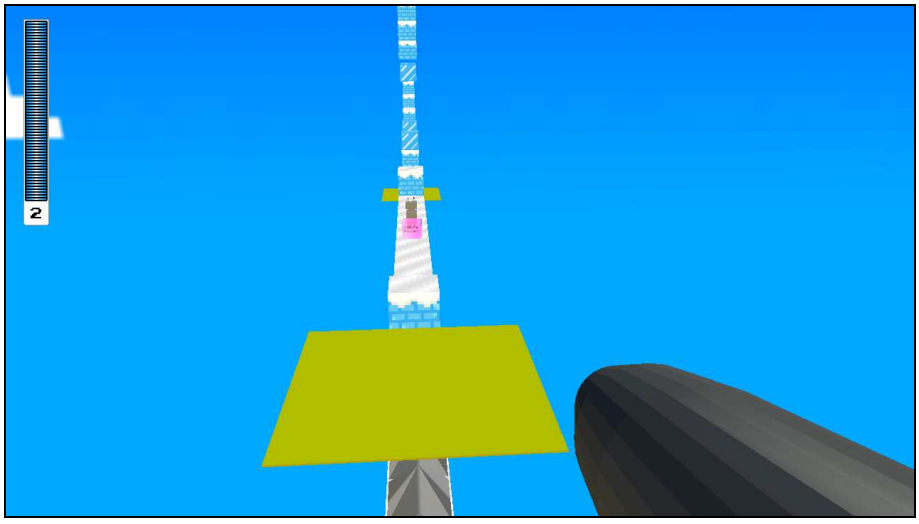

Figure 12: Riding an elevator.

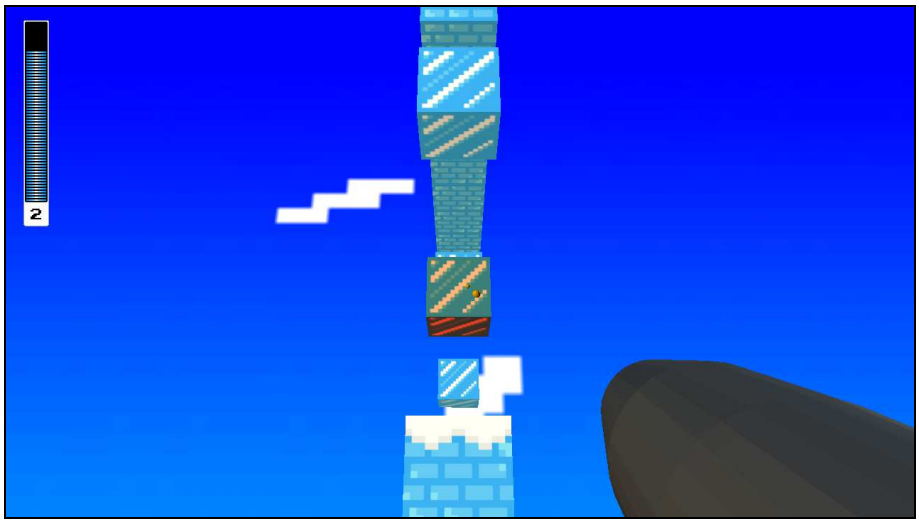

Figure 13: Moving blocks.

5. Discussion

5.1. Fulfilled Requirements

All the requirements that was listed at the beginning of the course were fulfilled. Submission of the game to Swedish Game Awards was made on May 31, consisting of a demo-level with about 5 minutes of gameplay.

A bitmap representing the level and a text-file for defining enemies and moving platforms were made early in the development. Physics, intersection-tests and response were the hardest and most time consuming part to implement. Enemies and some other objects were crudely modeled in Blender. Some models had material properties only, and others had textures as well. For shading, the built-in BasicEffect was used, decent, but enough to illuminate. A class for handling music and sound effects was created, although at this moment only sound effects are used in the game. Finally, a sky-box was defined to the scenery.

5.2. General Aims and Objectives and for the Programme

After this course I think I have shown that I possess a broad knowledge in the subject I studied. I could analyze and define solutions mostly by myself, but for some problems I needed to discuss the problem with my instructor to get a clear understanding. Regarding methods and theories I was able to mostly find the proper methods by myself. For problems where I couldn't find any method, I had to be creative and imagine a solution in my mind before starting to implement it. This worked a lot of times, but for some problems I needed to discuss it with my instructor.

For planning, I think I did a good job, during my entire education I have strived to always be a bit ahead, so I know I can relax more at the end of a course, in case something unexpected should happen. During this course, I felt stress sometimes, but only at a degree which made me focus more on the project. It was a pleasant course and the planning turned out very well, and approximate. Concerning the written and oral exam, I think I did a good job with the report itself. The oral exam could have been better though, it felt as I got the message out and with the help of the slides kept the audience interested. Although, I wasn't very present myself. But it was a good experience to being able to perform an oral exam for 30 minutes and I managed to get the presentation time for the slides very close to 20 minutes, which was recommended. I had gone trough the slides with the instructor before and got a lot of good feedback how to adjust the tempo of the presentation to retrieve a time that wasn't to short nor too long. In the future, if I'm going to have a similar oral exam, I think I will find it easier to achieve more eye contact with the audience.

During the entire education, I feel that I have gained a better ability to derive solutions with relevance to a particular problem. I'm very aware that I still lack a lot of experience, I noticed this already some weeks into this project, at the beginning of the course I thought I was ready. However, I gained a lot of experience just during these 10 weeks. Besides studying the courses throughout this programme, I have spent a lot of free-time learning how to make games, that experience gained me a lot during this degree project. Without it, I'm not sure I've could have finished this at all. This is just the beginning, if I look relative on problems that have occurred previously during my education, a lot of them seems trivial now. In a couple of years I think I can call myself a fulfilled game programmer, maybe even designer.

5.3. Future Improvements

I'm pretty pleased with what I have accomplished so far. After I got the intersection response to work correctly and had added some sound effects, the game started to feel like a game. I will continue to develop this game on my own after graduation. Unfortunately, I can't do that all the time, since I need to study or work to get money. But, I will devote a lot of spare-time for the development of the game. In the end, I want a product that feels professional and flawless and try to submit it to *Xbox Live Indie Games* [3].

With everything implemented so far there are a lot of things that could be improved. For example, the textures of objects and better looking models. I'm going to learn Blender better and create more detailed models that are also animated. Also, particle-systems could be used for visual effects, e.g. snow. Terrain in the background and around the level could be used for more variation in the environment. There are also a lot of code that can be optimized. An example of this is the bounding-boxes for the cubes, these are now only merged horizontally, but can also be merged vertically for reducing the number of bounding-boxes created.

Another thing that I will learn, but haven't used yet, is *HLSL (High Level Shader Language)*. Right now, I'm using the BasicEffect-class, a default shader in XNA. With HLSL it is possible to render the environment more freely, and it is more flexibly to separate the code for the shading from the rest of the code.

Of course, there must be music in the game as well. Throughout my life, I've always been inspired by video game music, especially from older games when hardware was an issue and the composers were forced to create more melodic tunes rather than atmospheric music. I've been composing and producing music for about 10 years now, and I would say I get better all the time, not a professional, but enough to compose music that people like. By creating music for us in the game, will add a new dimension for composing music for me, I think.

Right now, the only way to make a level in the game is trough a bitmap, it's a fast and easy way to create or edit a level, but not so efficient when more and more objects and textures are added to the game. Because one color-value must be used for each different texture and object. Therefore, I will create a level-editor, where it is easy to assign the texture to use for a certain cube and also for adding enemies, platforms and other objects.

6. References

[1] *Swedish Game Awards*
Retrieved: 2012-02-22
http://gameawards.se/

[2] *XNA Reference*
Retrieved: 2012-04-09
URL: http://msdn.microsoft.com/en-us/library/bb200104.aspx

[3] *App Hub – FAQ*
Retrieved: 2012-04-16
URL: http://create.msdn.com/en-US/home/faq/xbox_live_indie_games#xboxfaq-registration

[4] *GNU Image Manipulation Program: User Manual*
Retrieved: 2012-05-10
URL: http://docs.gimp.org/2.6/en/

[5] *Blender: User Manual*
Retrieved: 2012-05-10
URL: http://wiki.blender.org/index.php/Doc:2.6/Manual

[6] *XNA Game Studio Connect*
Retrieved: 2012-04-08
URL: http://msdn.microsoft.com/en-us/library/bb975643.aspx

[7] *Skybox Tutorial*
Retrieved: 2012-05-09
URL: http://sidvind.com/wiki/Skybox_tutorial

[8] Reed, Aaron, *Learning XNA 4.0*. Farnham, UK: O'Reilly, 2011 – ISBN-10: 1449394620, ISBN-13: 978-1449394622

[9] *XNA 4.0 First Person Camera Demo*
Retrieved: 2012-05-09
URL: http://www.dhpoware.com/demos/xnaFirstPersonCamera.html

[10] Buckland, Mat, *Programming Game AI by Example*. Plano, Tex.: Wordware, 2004 – ISBN-10: 1556220782, ISBN-13: 978-1556220784

[11] *Equations of Motion*
Retrieved: 2012-05-10
URL: http://physics.info/motion-equations/

[12] *XACT*
Retrieved: 2012-05-16
URL: http://msdn.microsoft.com/en-us/library/ff827590.aspx

[13] *Audacity*
Retrieved: 2012-05-14
URL: http://audacity.sourceforge.net/

[14] *FL Studio 10*
Retrieved: 2012-05-16
URL: http://www.image-line.com/documents/flstudio.html

[15] *Attribution 3.0*
Retrieved: 2012-05-16
URL: http://creativecommons.org/licenses/by/3.0/

[16] *Creative Commons*
Retrieved: 2012-05-16
URL: http://creativecommons.org/about